307

MANAGEMENT OF AN INTELLIGENT DATA OBJECT

by

DOROTHY MONTGOMERY GANTT

B.S., University of South Carolina, 1972

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:

Major Professor

LD
2669
.R4
CMSC
1987
G36
c. 2

A11207 303010

CONTENTS

1. CHAPTER ONE - INTRODUCTION..... 2
1.1 Forms Management Research..... 3
1.2 Why Electronic Forms Based OAS?..... 4
1.3 Actor-Based Programming Systems..... 4
1.4 Abstract Data Types..... 6
1.5 Form Languages..... 7
1.5.1 Form Fields..... 9
1.5.2 Form Abstraction..... 10
1.5.3 Form Access Rights..... 10
1.5.4 Form Operations..... 12
1.6 OAS Specification Languages/Models..... 12
1.7 Active Electronic Message Systems..... 14

2. CHAPTER TWO - AN INTELLIGENT DATA OBJECT MANAGEMENT SYSTEM..... 19
2.1 Creating The IDO..... 21
2.2 IDO Procedures..... 22
2.2.1 Routing..... 22
2.2.2 Processing..... 23
2.2.3 Error Handling..... 25
2.2.4 History Log..... 25
2.3 Node Manager..... 26

3. CHAPTER THREE - A NODE MANAGER MODEL..... 27
3.1 Requirements..... 27
3.2 Design..... 27

4. CHAPTER FOUR - IMPLEMENTATION AND SIMULATION OF A NODE MANAGER..... 35
4.1 Introduction..... 35
4.2 Simulation Design..... 36
4.3 Implementation Design..... 40

5. CHAPTER FIVE - CONCLUSIONS..... 43

6. CHAPTER SIX - EXTENSIONS..... 45

APPENDIX I - MANUAL PAGES..... 47

APPENDIX II - SOURCE CODE STRUCTURE CHART..... 82

APPENDIX III - SOURCE CODE LISTINGS..... 86

APPENDIX IV - SAMPLE LOG FILES..... 125

BIBLIOGRAPHY..... 131

ACKNOWLEDGEMENTS..... 135

LIST OF FIGURES

Figure 1. AN INTELLIGENT DATA OBJECT MANAGEMENT SYSTEM..... 20

Figure 2. IMPLEMENTATION PROCESS STRUCTURE..... 36

Figure 3. idosav DIRECTORY STRUCTURE..... 37

Figure 4. IMPLEMENTATION DIRECTORY STRUCTURE..... 39

1. CHAPTER ONE - INTRODUCTION

Currently in computer science research and in industry, there is a great deal of attention directed to automation of the office environment [BAU80, FIK81]. This attention is centered on development of systems and facilities to aid the office worker in the more basic aspects of his or her job, to improve productivity, and to stem the tide of rising costs [ZIS78].

Office information is managed by several office functions: text editing, forms editing, organizing, preparing, filing, copying, transforming, analyzing, storing, and transmission [ELL80, LAD80]. These office functions have been or can be automated individually. However, what is difficult and appears to be the direction taken in the field of automation is to integrate these functions such that the user interface will be minimal and less complex; the information flow control will be enhanced; and the office functions will become more efficient [ELL80].

This report will review the research in Office Automation Systems, particularly the research of electronic forms management. The concept of an "intelligent data object" will be further explored, with the main thrust of the report on the implementation and simulation of an intelligent data object node manager.

1.1 Forms Management Research

One area of Office Automation Systems (OAS) where there has been much research activity is in forms management [GEH82, LAD80]. Two types of integration must occur when referring to forms management. The first type is integration of the different facilities (media and/or machines) used; which may consist of typing, copying, telephones, printers, etc. The second type is integration of the different services and/or interfaces which are provided by different individuals, organizations, companies, etc. Both types of integration require coordination of technical and human interface levels. Technical and human interface levels include voice communications, graphics, query languages, word processing, data base management, phototypesetting, electronic mail, and others within a specific OAS [TSI82.1].

Electronic forms enable this integration of facilities and services/interfaces. Normally the word "form" invokes to ones mind a paper entity called a "business form", but forms can also assume a role which is greatly expanded to the point of appearing as an entirely different entity in the OAS environment. Electronic forms can be an abstraction and generalization of business paper forms, which can be created, destroyed, changed, mailed, and located. An automated electronic form can be coupled with automated form procedures and allow the form to assume some intelligence of its own [TSI82.1].

1.2 Why Electronic Forms Based OAS?

Electronic forms are a natural basis for an OAS. Utilized as a logical entity (an abstract data type that is a repository of procedures and data) and for user interface (allowing movement, modification, and manipulation) electronic forms have the following benefits versus other systems [GEH82]:

- logically related data is treated as an entity
- the properties of paper forms can be retained and thus the user is more comfortable with the interface
- tracing is more efficient - the current processing state can be determined readily upon a user inquiry
- particular form types can be tightly coupled to specific interaction of specific users
- particular forms can have specified routing associated, which is performed automatically
- high-level protocols for information communication can be predetermined
- if transitioning from a manual office system to an automated office system, the impact on office personnel is minimized
- paper copies can be generated upon demand or automatically
- user interface is improved by structured input and global display of data

1.3 Actor-Based Programming Systems

Electronic forms can incorporate both text and data. With text we can specify form procedures and management/processing of data as well as the actual data itself. Text is what permits the form intelligence to exist. As a result of having an intelligent form, one then can have an OAS where the intelligent forms can

communicate via messages among themselves and with their environment without any or with a minimum of human interaction. The message environment is viewed as a distributed system. This system has a network of processing nodes which can have movement between these various nodes; this system is analogous to the functional components of a business organization's objects e.g., mail, invoices, purchase orders, personnel forms.

The emphasis in programming languages is changing from procedures which act on passive data to active data processing messages. The actor model supports this new perspective. The naming architecture of an actor-based programming system provides a uniform environment in which distributed applications can be automated in a highly modular and efficient manner [BYR82].

Electronic forms are similar to an "actor-based programming system" where [BYR82, HEW77]:

- actors are the objects in the system which execute and communicate via scripts
- actors have a memory
- actors communicate by messages, which in themselves are actors
- actors execute asynchronously after being triggered by a message event
- actors can include: effects of parallelism, instances of monitors, envelopes, and serializers
- actors are highly modular
- actors have well-defined interfaces

- actors can be created by other actors

1.4 Abstract Data Types

An actor-based programming environment enables the utilization of a data type specification (or abstract data type) that is independent of its eventual implementation. An abstract data type has been defined to be a class of objects which is completely characterized by the operations which may be performed on objects of that class.

Utilization of abstract data types makes the practice of structured programming more understandable by providing a means in which the abstractions uncovered during the course of program design could be expressed more naturally.

Through the use of abstract data types one obtains modularity, an increase in maintainability, and the possibility for a proof of correctness, which results in an improvement in program quality [LIS74].

The behavior of the abstract object becomes the concern of the user of the abstract data type and the irrelevant details of storage representation and implementation of operations are hidden. This allows the user to assume a "what they do" and not "how they are implemented" orientation. Thus allowing concentration on the problem to be solved and the exclusion of the irrelevant details [LIS74, GUT76].

1.5 Form Languages

Coupling form definition languages and form manipulation languages with the concepts found in abstract data types proves to be an excellent, although not a "perfect" marriage.

Several form definition/manipulation languages exist in research and in practical applications of which some are listed below:

- OFS (Office Form System) - An integrated form management system which combines three types of activities: desk activity, coordination activity, and mail activity. [TSI80]

- QBE/OBE (Query-By-Example/Office Procedures By Example) - QBE "combines subsets of computer domains as word processing (including edit and format), data processing, electronic mail, report writing, graphics, security features, and application developments within a single interface." OBE "combines subsets of computer domains as word processing (including edit and format), data processing, electronic mail, report writing, graphics, security features, and application developments within a single interface." [ZLO81]

- ODIN (Online Data Integrity System) - A set of software tools, which run under UNIX, developed to automate the construction of electronic form entry, processing, and retrieval procedures for a relational database. It utilizes a language which defines the logical structure of forms, the integrity constraints that each form has, and data mapping facilities. ODIN defines the operations and the forms that a user is permitted to view. The integrity constraints to be enforced are defined on a field, form, and multiform basis.

  Forms may be entered interactively using a sophisticated full screen forms editor, or in large quantities entered from programs or data files using a bulk form entry tool. ODIN permits real time modification of data to an on-line database, with programs which verify the validity and integrity of the data prior to mapping the changes. A HELP facility is available to the user from the forms editor. A

report generator is also available which permits sorted output. [FER82, DIP83]

- TAXIS - A language for the design of interactive information systems. It offers relational database management facilities, a means of specifying semantic integrity constraints, and an exception-handling mechanism, integrated into a single language through the concepts of "class", "property", and the "IS-A (generalization) relationship". [MYL80]

- ODYSSEY - A representation and use of task domain knowledge to assist with the acquisition of data in an office information system. Utilizes "frame-oriented" style of programming to design and implement the tool which combines "frame-structured" knowledge representation and "object oriented" programming. [FIK81]

- OFFICETALK-D - A language which allows multi-computer distribution and sharing of control within an office environment. OFFICETALK-D "allows for the flexible manipulation of electronic forms on the display screen of users and helps to coordinate and control the flow of forms between user workstations." "Novel facilities implemented in the system include distributed schedules, dispatchers, office observer workstations, alerters, a data dictionary synthesizer, change agents, and on-line office modeling, simulation and design facilities." [ELL82]

Four aspects of form properties, which must be considered when performing form definitions, are [GEH83, GEH82, TSI82.1]:

- fields - blank spaces in forms where required or requested information is to be inserted

- abstractions - encapsulation within a form of the information necessary and relevant to performing one or more tasks

- access rights - control of the user's authority to create, destroy, copy, modify, review, or fill out forms

- operations - which operation types (e.g., create, delete, modify, destroy, print, etc.) are permitted for specific form types

1.5.1 **Form Fields** Fields in electronic forms can be more complex than paper form fields and can have more variety of field types. The complexity of fields and the variety of the fields are determined by the electronic forms designer.

Some types of fields that can be provided are [GEH82]:

- personalized - data filled in automatically from the user profile and system variables, e.g., name, workstation number, date
- required - data which must be entered by the user when filling out the form
- unchangeable - data entry is optional but cannot be changed once entered
- unrestricted - data entry is optional and can be changed at any time
- virtual - this field is filled in automatically according to some specified computation using computation rules, such as procedures
- tag - the value of this field determines the selection of the appropriate variant from a set of variants
- variant - the filling of these fields is prevented until the corresponding tag field is filled in, as only one variant is selected per tag field value
- dependent - these fields can be filled with data that satisfy some constraints
- ordered - can be filled only after some other fields have been filled
- lock - filling up of this field results in certain other fields being protected from modification
- conditional - this field is filled by the user on the condition that information is not available in the associated database or is to be changed (allows for default initial values)
- invisible - this field will be invisible to users who do not have access rights to read or

update the field

- variable length - the amount of text that can be filled is unlimited
- repeated - the number of instances of this field would depend upon the needs of the user

1.5.2 **Form Abstraction** Electronic forms can be used as a tool for abstraction, where the information necessary and relevant to performing one or more tasks is encapsulated [GEH82].

Within an OAS exists the need to specify general office procedures which can be invoked according to prespecified conditions and require minimum user intervention (if any).

Procedures should be initiated when certain preconditions are met; perform the action; and then test for some post-condition; with a set of "failure" operations when either pre -or- post-conditions are not met [TSI82.1].

Design decisions need to be made as to what capabilities should be included in the procedure specification plan for implementation, and what the implementation environment is to be. These decisions can be made using the assistance of one of the office information/automation system modeling tools that are available.

1.5.3 **Form Access Rights** Who should be able to create, destroy, copy, modify, or fill out forms?

Access rights must be determined based upon the data being accessed, the operation being performed, and the "need to know". Access of forms must be controlled in a manner consistent with the OAS needs.

There is disagreement as to the implementation of access rights. Some say that the workstation, not the user, should control the access rights [TSI82.1], while others advocate that users access rights be associated with the forms themselves. It is suggested that the association of users access with forms is a more flexible system in that [GEH82]:

- access rights assigned to a user will depend upon the function and rank of the user in the organization
- form access rights govern access at the level of fields and the operations that can be performed on the forms
- allowance can be made to permit some users to have access rights to change access rights of others
- delegation of subsets of access rights can be made by one user to another user
- association of access rights can be made to the owner of the program itself
- agreements can be made between OAS's to enforce access rights when a form is transmitted between OAS's

Restriction of access rights by workstation appears to imply a limited delegation of authority environment. If a particular type of station can "only" perform specific queries about forms, this method of controlling access seems to be somewhat inflexible.

1.5.4 **Form Operations** The form definition specifies which users can perform which of the operations that apply to a particular form instance. Form operations may include [GEH83, TSI82.1]:

- creation of form instances
- entry of attribute values in forms, either manually or automatically
- storing forms
- copy unofficially a form and provide a new identification number
- copy officially a form and exercise extreme control when doing this
- modification of attribute values
- deletion of form instances
- routing of form instances or form values between stations
- distributed query capability, where (upon occasion) a user should be permitted to query data that are distributed in many stations)

1.6 OAS Specification Languages/Models

Office specification languages are quite important in the formal description, analysis, and planning process of an OAS. They can be utilized as valuable tools in representation, documentation, analysis, design, and evaluation of an OAS. The diverse aspects of offices lead to different modeling approaches. Some examples follow.

The OFFIS system [KON82] facilitates an interactive and iterative analysis and design process of providing the planner/designer of the automated office with a flexible method of documenting and analyzing system features and constraints. This model consists of

a framework representation which includes definitions of objects, attributes, and relations:

- office functions (clerical, monitoring/tracking, and control)
- information flow
- activity models
- hierarchy structures

The form flow model (FFM) regards an office as a network of stations through which forms flow [LAD80]. The forms are conserved in the network; they are neither created nor destroyed, only transformed. The network is deterministic, isolated from all factors external to it, and the network is memoryless.

An automated workflow central model developed by Baumann and Coop [BAU80] provides an office process model which isolates workflow control from the individual personal processes that constitute the office process. This model considers only a single control span within the office organizational hierarchy. Multiple spans of control in the office hierarchy are obtained by inter connection of the independent workflow control models, and decomposing work definitions in accordance with organizational charters.

ABL (Alternative Based Language) is a modeling office information systems language which provides for modularity, the description of parallelisms, and is open to stepwise refinement. [LEB82] ABL also permits the user to interact with the model in a variety of ways by allowing modification at the specification, design, and

implementation stages.

OSL (Office Procedure Specification Language) [HAM80] deals more directly with office functions. OSL permits document description and patterns of communication. Language structure and syntax are tightly coupled to an augmented Petri Net formalism.

ICN (Information Control Net) provides "...a comprehensive description of activities, tests the underlying office descriptions for certain flows and inconsistencies, to quantify certain aspects of office information flow, and to suggest possible office restructuring permutations." [ELL80]

As an example of an ICN application, Cook has utilized the ICN model to perform a particular type of transformation, streamlining. [COO80] This technique of streamlining reduces the ICN model of a procedure to the necessary information flow and elementary information-processing. Streamlining can highlight origin and destination of procedure information, permitting the modeler to vary the routing of information. Streamlining also shows information-processing needs, activity-by-activity, such that evaluation or change of the original procedure can be implemented.

1.7 Active Electronic Message Systems

Stefferud and McHugh say that office systems should permit the receipt of correspondence, as well as other information through some sort of information transfer processes, and then merge this

with other information which has been retrieved from files. The combined information should be transferred to others by a "third party delivery system." [STE81]

Basic electronic mail systems transmit messages (a unit of communication) between originators and recipients. A computer-based message system (CBMS) arbitrates the specific systems communication and provides facilities for users to read and create messages.

MSG was one of the early computer-based message systems. It was developed for the ARPANet environment to facilitate the ability to forward messages to other people and to reply to mail. Utilizing commands and user profiles, the aspects of MSG's behavior could be altered. [VIT81.1]

Research and development efforts have treated messages as passive information (text or facsimile) while there exists a growing demand for very general and dynamic facilities melded within the traditional message processing functions. [TAR81, VIT81]

These desired "dynamic facilities" include [VIT81]:

- "a mechanism which resides between a user and his incoming mail, and performs some amount of preliminary processing"
- "active message capabilities (i.e. messages as executable object or procedures) which allow messages to perform some actions on their own"

- "a modified command structure and user interface which allows users to add their own functionality to the system, create their own interfaces, or modify existing functionality or interfaces"

The R2D2 (Research-to-Development Tool for Message Processing) system was developed to implement an "active message" system. An active message is a message that contains a procedure (i.e. a set of instructions triggered by some event). Active messages comprehend what processing is permitted and prohibit certain processes to be performed. An active message is capable of self-modification; it can execute changes upon itself during its life-cycle. [VIT81]

Active message systems permit the integration of the two office functions which need to be merged: electronic mail (e.g., message systems based on networks) and electronic filing (e.g., database management systems based on computers). [TSI81]

Tsichritzis developed OFS (Office Forms System) [TSI81] which integrated three types of office procedures: desk activity, mail activity, and coordination activity. Desk activity was the office procedure which specified those actions at a local station which were triggered by specific conditions at the station. Mail activity was the office procedure which automatically routed the mail. Coordination activity was the office procedure which provided specific form types and conditions which initiated specific procedures and then performed notification to one or more

other stations about the event. OFS produced an integrated form management system.

The "Worm" programs developed by Shock and Hupp were an experiment in programs or computations that could move from machine to machine, confiscating resources as needed, and replicating itself whenever the need required. [SHO82]

Another prototype system was developed by Tsichritzis which managed messages as typed objects sent from station to station. The system enabled the user to find and query information from these messages. The system also permitted specification of automatic procedures triggered by message presence and the procedures could manipulate the messages. [TSI82]

With the mushrooming utilization of electronic messaging and mail systems, increasing attention is being directed to the developing problems of naming, delivery, processing, addressing, and routing. [GAR81, TSI84] Schicker proposed [SCH82] that a set of attributes (names, postal addresses, company affiliation, etc.) be mapped into an identifier which would designate a single "originator or recipient" (O/R) of mail. The mail system would convert the O/R identifier into an address which would be utilized by the mail system for forwarding and delivery. Methods were also developed to permit the mail system to deliver mail correctly even though not all of the accurate information was available. Kerr also recognized the fact that the rapid increase of electronic mail

systems would defeat their usefulness unless methods of interconnection were agreed to and implemented. [KER81]

In 1981, a draft of the National Bureau of Standards Message Format Standard for the format of messages on computer based message systems was developed. [DEU81] While the draft did not completely specify the representation of all parts of a message, it did illustrate the necessity of the development of standards for successful, efficient, and productive interconnection of different computer based message systems (CBMS). In addition standards have been and are being developed within CCITT (International Telephone and Telegraph Consultative Committee) to standardize service definitions (interconnection standards). [SCH81]

There has existed message systems which transmitted and received messages but did not administer the information contained in the message; and there has existed database management systems which managed the information of a global database but did not have the concept of address transmission or receipt. Now there exists the concept and capability of having both merged together as an "active electronic message system."

2. CHAPTER TWO - AN INTELLIGENT DATA OBJECT MANAGEMENT SYSTEM

While current active electronic mail systems are primarily text-oriented, there exists the capability for messages to contain text and information that is not represented as just text or data structures but graphics, facsimile, digitally encoded voice, and/or procedures.

Hogg views an "intelligent message" as an active program which can have a dialogue with the recipient and can route itself based upon pre- and/or post-conditions. The routing is not just to one recipient but can be to multiple recipients that can "return" responses back to the sender when the processing is terminated. [HOG84]

A model was developed by McBride and Unger [MCB83] of an intelligent form "...which can be used to depict the control and information flow of a job in a distributed processing environment. The basic technique that is used employs individual Petri nets to describe each of the procedures that a unit of work must invoke." Their paper [MCB83] on an "intelligent data object" (IDO) was the inducement for this report.

The IDO may be viewed as an intelligent abstract data type consisting of text (data structures), and procedures (routing/navigation commands, processing, error handling, and a history log) encapsulated in an envelope called an electronic form

which is routed as an active electronic message (like an actor based system) from node/station to node/station, where the processing of the IDO may be caused by a trigger of pre- and post-conditions.

A node/station (hereafter referred to as a node) is that domain or location where the environment exists or will be established for an IDOs processing to occur. It is the point at which responsibility for a message is transferred within a distributed network system to another "node". Each node has a unique global identifier to distinguish itself from all other nodes. Each user of the OAS system operates at a single node, where IDO processing occurs. The node has within it resident database(s), resident procedures, resident commands, resident error handling, and a node manager.

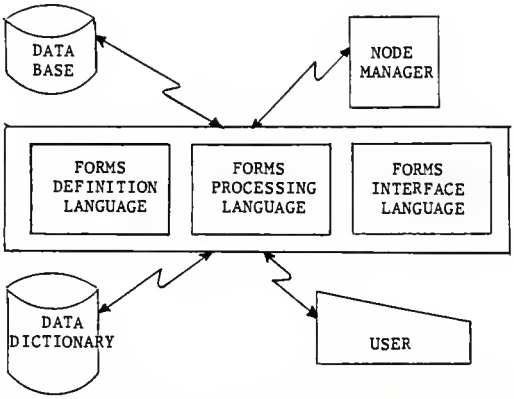


Figure 1. AN INTELLIGENT DATA OBJECT MANAGEMENT SYSTEM

2.1 Creating The IDO

Using a forms definition language, an IDO is created with a unique identifier to the OAS. The forms definition language should be menu driven, where the form is displayed on a CRT (Cathode Ray Terminal) screen through a screen manager and/or forms display language. The form should prompt the user on a field-by-field basis for required and optional information. It should step the user through each field; forcing the user to input required fields; requesting data for optional fields; and inputing default values as necessary. As the user is inputting the data into the form, the forms definition language will perform integrity checking (e.g., range checking, type checking, and consistency checking on a per field and per form basis). An example of this inter- and intra-form checking follows:

Example Of: intra-form checks

Given Form X - field 10 has a value of 567
- by predetermined rules -
Field 25 "must" contain numeric data and
have a minimum value of 5 and a maximum
value of 20

Example Of: inter-form checks

Given Form X - field 23 has a value of xyz
- by predetermined rules -
field 5 in Form Y "must" have a value
of abc because there exists special
dependencies between the two forms (X and Y)
and their respective fields (Form X/field 23
and Form Y/field 5)

At form creation time, using the form definition language, the user specifies all fields: those fields which can be filled-in

automatically, fields which can have particular access rights, operations that may be performed on specific fields, fields that are optional or required, fields that are locked, etc. The forms definition language defines what operations are permitted on the completed form. Some operations performed on forms may be: create/add, delete, modify, query/read-only, copy, route/navigate, etc.. The forms definition language specifies access rights/security constraints for the form as a whole (not just the individual fields).

2.2 IDO Procedures

The forms processing language enables the creation of specific IDO procedures. These procedures consist of several types (e.g., routing/navigational, processing commands and routines, error handling, and history logging.

2.2.1 **Routing** The routing procedures enables the IDO to route itself within a distributed network. This routing can be between two or more terminals on a local environment or can be general navigation between 2-N stations in parallel or in series.

The form itself must be able to notify the system when the form (IDO) is ready to be moved to another location using some trigger. This self-initiated routing notification, as well as the identity of the recipient, can be one of the processing commands for routing purposes. During routing, the form must notify the sending and receiving node managers of its actions.

The routing procedures must be able to manage the situation if a message is lost (e.g., have a default routing routine where the message is re-routed or else action to represent itself again) or a message is delayed (e.g., have a specified timeout response period for acknowledgement of receipt and some sort of default action to be taken when timeout occurs).

The resident node manager will control the routing once the IDO has either initiated routing.

2.2.2 Processing In addition to the routing procedures, which gives the IDO some semblance of intelligence, the processing procedures represent the strength of the IDOs intelligence. Processing procedures can provide the IDO with the capability to make decisions about the next action to take place. The forms definition language will enable definition of what the pre- and post-conditions are for IDO form processing.

The IDO, using processing procedures, will have the ability to perform computations and various functions based on those computations. The IDO will be able to compute fields from existing fields (e.g., totals on an insurance form, extensions on a purchase order, inventory totals). Using fields within the IDO and retrieved data from other sources (including node resident databases) additional calculations can be made. The IDO calculation processing procedures will also contain information of:

- when (via triggers, pre- and/or post-conditions etc.) the

calculations are to take place

- how they are to take place
- when the requisite data is to be stored
- how the data is to be stored
- where the data is to be stored

Processing procedures also include providing the IDO with the self-processing capability to follow a sequence of commands/instructions which can be anything from stimulating database retrievals via a database retrieval language; form creation which can be of new IDO types or instances of current IDOs; form modification (e.g., add operators or operations to a form); form deletion; enable stimulation of self-initialization of a program at various points in the decision process; to form storage, etc.

The IDO must be able to detect by processing procedures when it is not receiving adequate attention from a user. If user response is expected within a specified amount of time during a particular process then the IDO must be able to respond accordingly via predetermined procedure.

Utilizing processing procedures and a database retrieval language (forms interface language), the IDO can retrieve data from memory

locations of either registers or from fields other then in databases (i.e., flatfiles).

2.2.3 **Error Handling** Error-handling procedures for IDO processing may be a part of the IDO itself, resident at the node, or a combination of both (where the IDO has specific error procedures for itself and the node has general error procedures for the node itself). Based upon the results of IDO procedure processing, commands or operations, the node manager will invoke specific error-handling procedures as necessary based upon the algorithms provided to the node manager.

2.2.4 **History Log** A history log is maintained by the node manager with what processing the IDO has done and where it has been. The resident node manager will need to use this history log which is part of the IDO for various functions.

This history information will be utilized for tracing the IDO when necessary (e.g., a lost IDO, a user query, or a re-routed IDO). Should errors occur in processing, the history log will provide invaluable assistance in debugging (assuming the error isn't in the history log processing).

This history log will travel as part of the IDO from node to node, with a copy kept at the originating node which passed the IDO to the next node. To insure that the IDO received at the new node is intact, the communications handler will verify the integrity of the IDO after receipt, and determine if the IDO needs to be

retransmitted. Once successful transmission has occurred to the next node, the originating node will then destroy his local copy of the history file after a predetermined period of time.

2.3 Node Manager

At each node there will exist a resident node manager which will provide centralized control for the node itself and for one or more IDOs that visit the node. This manager will handle receipt and transmittal of multiple IDOs. The manager will setup all mechanisms which establish the individual IDO environment based upon predetermined node routines and/or IDO processing procedures.

The manager will perform all processing necessary for the IDO including transmittal of the IDO to the next node(s) with an updated history log. A communications handler should verify the receipt and the integrity of the received IDO at the other node. Once an acknowledgement has been received from the receiving node, the manager will perform all cleanup operations necessary at the local node.

3. CHAPTER THREE - A NODE MANAGER MODEL

3.1 Requirements

The basic requirements for the node manager model are:

- Assembly of IDO components for routing
- Handle transmission and receipt of IDOs
- Schedule incoming IDOs to be processed
- Establish the environment and processes necessary for producing/creating an IDO
- Disassemble received IDOs
- Coordinate all IDO procedure processing
- Provide "hooks" for user interface
- Provide "some" security for the node and the IDO
- Provide status information upon request
- Cleanup
- Error handling

3.2 Design

The design of the node manager model (hereafter referred to as "node manager" or "manager") will encompass all of the above basic requirements with expansions of each one.

Each user of an OAS operates at a single node which has a unique global identifier distinguishing it from all other nodes. The system has the capability to generate a globally unique identifier for each IDO instance which the user has defined. This IDO identifier is permanently attached to the IDO and cannot be modified. Based on access rights a user can copy an existing IDO

any number of times; the new IDOs can be identical to the original except for the global unique identifier. The user can also destroy or modify any IDO.

Three major constraints exist for all managers of IDOs:

- The manager must execute the IDOs in a restricted environment to provide security. Since the IDO is a form whose contents are unknown to the recipient node, the opportunity exists for guerrilla actions. A user could send a time bomb in with his/her files which could override the local node environment, processing, etc.
- All IDO instances are physically unique (as determined by the global unique identifier)
- A particular node can only process IDOs within that node

After the user has created an IDO, the IDO "can" contain the following components (based on the type of IDO and scope of IDO processing):

- Pre- and/or post-conditions for processing procedures - General/ specific conditions which can be defined in terms that the manager checks the state of the IDO processing, initiates processing and/or routes accordingly. The pre- and post-conditions may specify which procedure processing can be performed by a particular form, node, database, etc. Timeout and error handling algorithms may also be pre- and/or post-conditions.
- Routing specifications - A distribution list, which need not be to just a single destination but several recipients can be sent a message at the same time.
- Local data
- A capabilities list describing the files/operations/commands that an IDO has access to and which may be carried out at the local node.

- A history of the transactions that have been executed; including a list of procedures processed, the time processing started, the time processing completed, the time the IDO was routed, a sequential list of nodes visited including the originator.

- Processing procedures/commands which specify: "what is to be done", "who is to do it", "how is it to be done", "where (environment) it is to be done", and "what happens next - after processing is completed".

The IDO is installed in the local node, regardless if the IDO is a result of having been transmitted to the local node or is the result of local node creation/modification.

Regardless if a user has logged in to the node, the local node manager will scan the node for the arrival of an IDO and then sleep for "x" period of time, wake up, scan, etc. The sleep time period is some algorithm which has been determined by the OAS administrator based upon the amount of job/processing traffic, resources, etc., on the local OAS. Once the manager detects the presence of an IDO it then parses it into its respective parts based on a predetermined naming convention. The manager determines if the IDO is at the proper node by looking at the node capabilities list to determine if the node has the files/operations/commands required for this IDO. If it is at the incorrect node, the IDO is re-routed. If it is the correct node, the manager proceeds to setup the working environment, directories, files, etc.

If the IDO did not originate at the local node, the communications handler will verify that the integrity of the IDO has remained intact. If the integrity is correct, acknowledgement of successful IDO receipt is sent to the originating manager by the local node manager. Should the integrity not be correct, the receiving manager transmits an acknowledgement of unsuccessful IDO receipt and waits for the re-transmittal of the IDO (or whatever procedure has been predetermined for this situation).

The first command is executed after all pre-conditions are met and IDO processing continues until either an error occurs or the entire set of procedure processing commands for that node is completed. After each individual procedure's processing is completed, verification of post-conditions are performed. Successful completion causes the history log to be updated and the next procedure command to be initiated. The manager will verify that all processing is completed, based on the contents of the command file and return codes. The manager performs cleanup on the IDO as prescribed by the IDO in one of its procedures.

Routing of the IDO will be determined when the IDO is designed. The routing design may be a combination of, or just one of, the following:

- by operation type requested - (e.g., retrieval of form Z from database N is performed at node W -or- payroll calculations are processed at node X)
- by form type - (e.g., type X form always goes to nodes A, B, and C)

- by origin of the form - (e.g., node A originated the form, therefore the form only has access to nodes J, K, L, and M)
- by predetermined distribution list
- based upon the results of processing (e.g., some sort of algorithm)
- based upon availability (e.g., each node is a bank teller and the IDO is a customer who wants the first available teller)
- pick at random - (e.g., don't know and don't care, keep routing until a match of commands is found or you run out of nodes)

Routing instructions for the manager "should be" very specific. The manager only acts as an intermediary between the local node and the receiving node(s) during the routing of the IDO. Once the routing for the IDO is determined the manager verifies that the destination node(s) exist. If the node(s) does not exist, then the manager must:

- use default or alternative routing procedures as prescribed by the IDO
- treat it as an error and perform specific IDO error handling routines

If the node does exist then the manager updates the history log as previously described. A procedure is performed on the IDO components to aggregate them together to form a message which is transmitted to the next node. Once the next node has received the IDO, the receiving manager sends the appropriate acknowledgements based on the condition of the received IDO. Should the IDO message need to be retransmitted, there are two methods of reissue which could be used. The "syntactic method" which embeds the entire original IDO message inside a new message and the "semantic method"

which modifies the existing message by adding new fields (i.e., information about reissue of the IDO message) to the original message prior to sending the message out again. The node manager will have to have procedures to provide for this type of circumstances or else a default procedure should be established as a standard node procedure.

A timeout should be established as to how long the local manager should wait to receive acknowledgments from the receiving manager, and what is to be done by the local manager when the timeout is exceeded. The local manager has a copy of the history log during the routing process in case of unsuccessful transmittal (e.g., lost IDO, garbaged IDO).

Once the IDO message has been successfully routed the manager then performs necessary cleanup processing. Cleanup procedures are resident node procedures fired by an IDO command and/or individual IDO processing procedures which are generated on a per IDO basis.

Two more routing scenarios need to be addressed. The first scenario is where the manager has to route the IDO during processing of the command list. A method must be devised to enable the next node to pick up command execution where the previous node left off. To accomplish this, the manager discharges all routing in the prescribed fashion after having set a pointer in the command list at the next executable command to be executed. The receiving manager after establishing the proper IDO environment starts

processing where the pointer is situated and moves it until it has to route the IDO. Prior to routing, the pointer is again moved to the next executable command and so forth.

The second scenario involves the situation where an IDO is a multi-part form. It is possible to partition the IDO form into sub-forms or copies; to allow them to be processed individually independent of one another; route the "parts" to other (multiple) nodes at the same time; and then merge them back together into a completed copy of the original form. The IDO must provide to the manager the ability to handle this situation via processing instructions, commands, and appropriate error handling routines.

The question arises as to how much intelligence the manager should have. For example - Is the manager expected to realize when an IDO is damaged during routing and determine the nature of the damage to the message as a whole; and then r onstruct the missing information based on what knowledge it has and what it has determined? What is the minimum amount of information that can be used to detect or repair a given amount of damage? Can a local node manager identify that a problem exists and be able to kill "lost" copies? This would require a significant amount of information to be exchanged.

Basically the manager is only as "smart" as the IDO determines that it should be. There are resident at each node predetermined procedures which allow the manager to perform some of the basic

node manager requirements, but the remaining requirements and any extensions of these requirements must be provided by the IDO itself.

4. CHAPTER FOUR - IMPLEMENTATION AND SIMULATION OF A NODE MANAGER

4.1 Introduction

Utilizing the requirements and design presented in Chapter Three of this report, a node manager model was implemented. The implementation concentrated on a top-down, high-level design, which utilized a very structured modular design. The lower level tasks of the IDOMS (e.g., the actual IDO processing procedures) were performed by token UNIX scripts (alias stubs).

The entire implementation was written in UNIX shell; developed on a VAX 11/780; written for two operating systems: Berkley UNIX 4.2 and AT&T UNIX 5.2.

To simulate and demonstrate the implementation feasibility of this project of independent processes at different nodes, usage was made of independent processes (IDOs) running in different directories (nodes with IDO directories underneath). Even though these directories reside on a common machine, with a common login, they communicate with each other through a single primitive (i.e., a clean function interface that does avoid side effects - there are no networking problems, no problems with multiple operating systems, no timing problems associated with a network (e.g., only one node sends - a node can either send or receive, etc.), no protocol problems, etc.), which is the only communication means between the processes in the multiple directory structure.

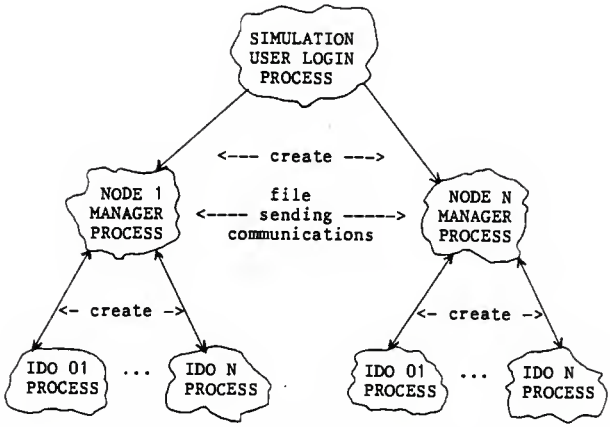


Figure 2. IMPLEMENTATION PROCESS STRUCTURE

4.2 Simulation Design

The simulation of the node manager implementation was designed to enable the user to execute the simulation with minimal effort. All simulation commands are easily identifiable by the "S" prefix. All commands which pertain to any IDO processing are prefixed with an "I". All commands which pertain to the node manager processing are prefixed with a "N". The commands which invoke IDO processing procedures are prefixed with a "J". One file (icmd) is used to list the IDO processing procedures (J-commands) to be executed by the node manager. The simulation user sets up the simulation by having a directory called "idosav", which acts as a shadow file for use during the simulation to create the node templates and the IDO templates. The "idosav" directory has the following structure:

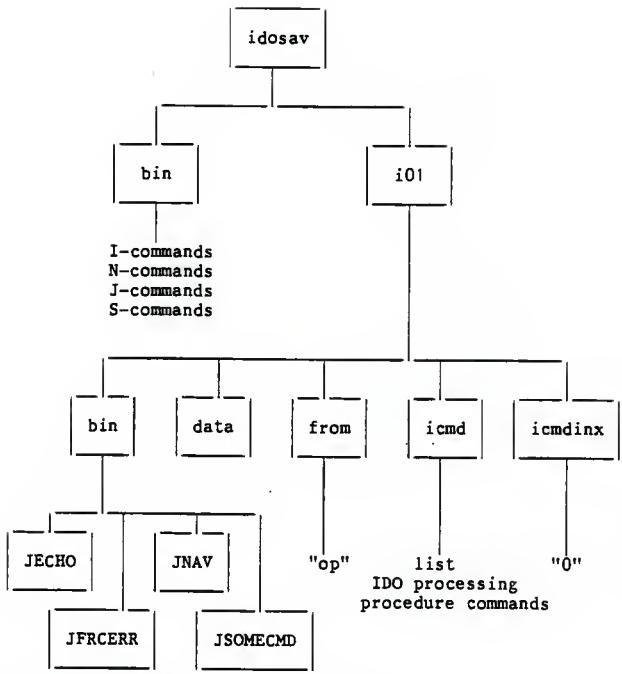


Figure 3. idosav DIRECTORY STRUCTURE

Within the "idosav" directory, simple IDO processing commands/procedures (in the directory "idosav/i01/bin/...") demonstrate the functionality of the node manager during the simulation; as well as the required "icmd" file which contains the sequence list by which these procedure commands are to be executed (in the directory "idosav/i01/icmd"). (REFERENCE APPENDIX III)

While in the "idosav/bin" directory, the user enters ". SRUNSIM", which starts the complete simulation with this one command. SRUNSIM will clean up the current local environment prior to establishing any of the needed simulation environment. A new "ido" directory is established, by calling SINIT, with three node directories of "a", "b", and "op" established underneath it. The "op" node will be utilized to originate and control the simulation.

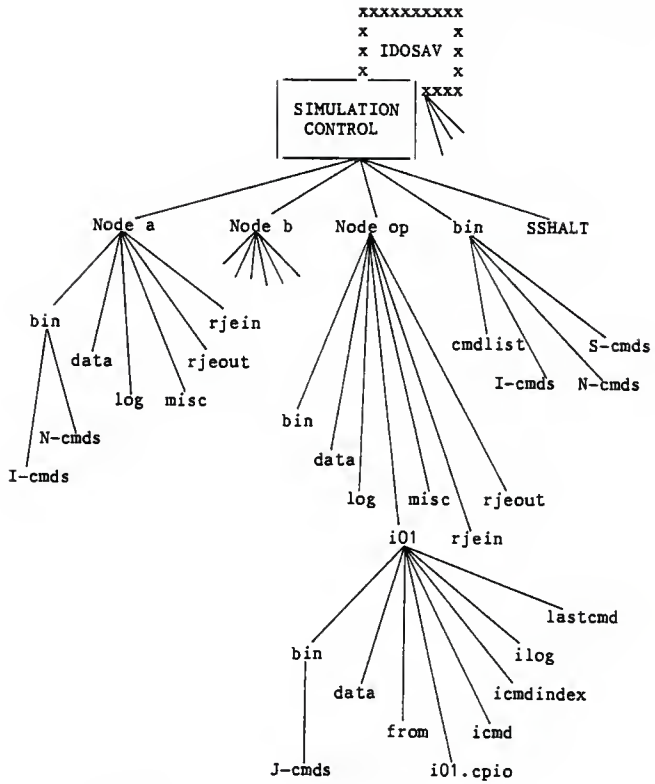


Figure 4. IMPLEMENTATION DIRECTORY STRUCTURE

The environment for each of these nodes is established when SETENV is called. The sample IDO (i01) used for the simulation is created under the "op" node. Then the IDO is routed to the first node (node "a") using SSEND. SSTARTUP is used to: wake up the local node manager's scheduler (NSCHED), as an independent background process;

process the IDO; and route the IDO as appropriate (based on the commands which are listed in the "icmd" file) to the next node (node "b") and then back to node "op". When all processing is completed, SCLEAN is called to clean-up all nodes of any files or directories.

During the simulation, the simulation user may "hook" into the running of the simulation by using one of three available commands:

- SRESUME - starts up a halted simulation
- SKILL - terminates all simulation node processes
- SHALT - temporarily halts the simulation

The user may also examine any of the log files for the IDO or the nodes at any time, either when the simulation is processing or when it has been halted. (REFERENCE APPENDIX III - SHALT and SRESUME, and APPENDIX IV)

4.3 Implementation Design

While the implementation and the simulation are coupled together for this project, the implementation can be disconnected and discussed here separately.

Assuming that the IDO environment and directory structures are in place and initialized (NINIT); that an IDO exists with some IDO procedures/commands (REFERENCE APPENDIX III - J-commands); and that a file exists listing the sequence these commands are to be processed (REFERENCE APPENDIX III - icmd), the IDO scenario would

be as follows.

The node manager loops (NSCHED) to look for waiting IDOs (ICKHALT); calls NMAIN if an IDO is found to schedule it for execution; and then waits for the next IDO. NMAIN performs two main tasks:

- schedules incoming IDOs waiting in the "rjein" directory to be processed by IPROC as an independent background process
- sends waiting outgoing files/IDOs to the next node through the "rjeout" directory.

NMAIN also makes appropriate entries in the node log (NLOG) for any of the tasks it performs. (REFERENCE APPENDIX III and APPENDIX IV)

Once IPROC is called, the IDO is disassembled into its components (IEXPND). An acknowledgement is sent (ISEND) to the originating node that the IDO was received. IJOB is then called to loop through and index (INCRF) the "icmd" file to determine the next IDO processing procedure command to be executed. IVALID is called to check the validity of the current command with the node command capability list. Each valid procedure is executed until no more procedure commands exist in the "icmd" file for this node. A record of the processing of the IDO procedures is entered in the IDO log file (ILOG). (REFERENCE APPENDIX III and APPENDIX IV)

Periodically the node manager will call ICKHALT to check to see if the user interface has halted simulation (for whatever reason) and take appropriate action as required. Once all procedures are

processed for the local node, IBUILD is called to assemble the IDO components for routing. ISEND will send the IDO to the next node through the "rjeout" directory and will wait for an acknowledgement (IWACK) from the receiving node. If the acknowledgement is not received in sufficient time, a timeout message is recorded in ILOG. Once again, during all processing, ILOG.(the IDO log file) and NLOG (the node log file) are provided with appropriate processing messages by each processing procedure. A sample node log file and a sample IDO log file obtained during an actual simulation execution are shown in reference APPENDIX IV.

5. CHAPTER FIVE - CONCLUSIONS

While this particular simulated implementation of a node manager model was not performed across multiple machines, it did demonstrate the functionality of a node manager, and thus it "appears" that the concept of an IDOMS node manager could perform across multiple machines.

During the implementation, each node manager did establish the local environment and processes necessary for processing an IDO. The components of an IDO were assembled for routing into an electronic message. The IDO was transmitted and received at "different" nodes. Acknowledgement was made by the receiving node to the originating node of successful transmission; demonstrating that communications were established between nodes. Each node manager did schedule the incoming IDO to be processed. The received IDO was disassembled by the node manager back into its individual components and the IDO procedure commands were processed in the correct order by the local node manager.

Minimum hooks were provided to the user to interface with the simulation. The user could halt, kill, or resume the IDO processing upon demand. The user could edit the individual processing procedures and the IDO command list file. The user could access the IDO log file and the individual node log file for status information.

Error processing was minimal. Security was provided not as part of the implementation itself, but only through the UNIX environment (e.g., set user id permission via CHMOD command).

Several UNIX scripts (alias stubs) were provided just to demonstrate the functionality of the implementation, its expandability, and its flexibility:

- the user could add or delete functions at each node with "no" or "minimal" impact
- it was demonstrated that IDO procedures could contain anything with "no" impact or "minimal" impact on implementation

With respect to the IDOMS project in general, it is felt that the "most" important conclusion obtained by the author of this report is that:

- by virtue of the IDO carrying with it its own intelligence, the IDOMS is "totally flexible" and could be "quite powerful".

6. CHAPTER SIX - EXTENSIONS

While the basic functionality of the node manager was implemented, the author of this report acknowledges that there are several expansions which could be made to produce a more robust implementation of the node manager model.

Security should be provided, where the manager can execute the IDO in a restricted environment or as some sort of secure buffer interface that exists between the user and the IDO. Perhaps the IDO itself could be encrypted during transmittal to prevent sabotage.

Performance has not been discussed at all in this report. The performance of the IDOMS as well as the specific performance of the node manager should be considered. For example:

- How many IDOs can be handled by the system at any one time?
- What kind of throughput should be expected or imposed upon the manager when processing IDO procedures?
- How often and at what rate should the manager schedule IDOs?

After development of the node manager implementation, the author felt that a "communications handler" should exist to facilitate the transmitting of IDOs between nodes. This communications handler would provide verification at the node as to the integrity of the received IDO. This handler should also provide for initiating re-

transmittal of IDOs when appropriate (e.g., when the IDO is "lost" during transmission; when the integrity of the IDO was verified and found to be unsound).

Although the basic error handling functionality was demonstrated, the error handling was very primative for this implementation. Much more robustness needs to be provided in all areas where any error handling occurs.

More sophisticated hooks need to be provided for user interface to the node manager as well as to the IDO. In addition to these "hooks", the interface between the manager and the forms interface language by the user should be further developed (e.g., to enable the IDO itself to obtain data through the forms interface language and the node manager, from the user --- facilitating a real-time user interface between the IDO and the user).

Procedures need to be developed for the manager to be able to handle duplicate IDO requests and duplicate acknowledgements.

While it is not the manager's function, there needs to be developed an interface and the procedures for the manager to be able to recompile an IDO prior to processing the IDO at the local node should that node have object code compatibility problems.

APPENDIX I - MANUAL PAGES

This appendix contains manual pages for the various commands and files used in the simulation of the "Management Of An Intelligent Data Object" implementation.

NOTE: All commands which begin with:

I = are IDO processing related commands resident at each node

J = are commands which simulate IDO processing procedures carried along by the IDO itself

N = are node related commands

S = are simulation related commands

Command Name and Abstract:

IACK - Sends acknowledgement of a received file.

Command Format:

IACK

Command Description:

This IDO command will determine which node sent the IDO; format the acknowledgement message; transmit the acknowledgement (ISEND); and make an appropriate entry into the log file (ILOG).

Command Name and Abstract:

IBUILD - Creates the IDO file

Command Format:

IBUILD

Command Description:

This IDO command will assemble any IDO conponents which are to be a part of the IDO into a CPIO format. Duplicate named IDO components are removed. The name of the sending node is attached to the IDO and the CPIO file is moved to the correct directory for processing.

Command Name and Abstract:

ICKHALT - Checks to see if simulation has been halted, and determines what type of halt occurred.

Command Format:

ICKHALT

Command Description:

This IDO command will check to see if the simulation has been halted for one of the following three reasons:

HALT - Simulation halted. Manager sleeps for 10 seconds and then continues.

KILL - Simulation terminated. KILL will force return of the calling command.

RUN - Execution will be resumed.

Command Name and Abstract:

ICLEAN - Performs cleanup activities on the IDO directory.

Command Format:

ICLEAN

Command Description:

This IDO command will perform basic cleanup activities on the IDO directories by:

- making an entry in the log file (ILOG)
- saving the error files
- removing all the IDO files

The simulation user creates the IDO ICMD file, which normally contains a sequential IDO processing command list. This file contains the "J" commands which are used for the purposes of simulating this particular IDO implementation.

The "J" commands used to simulate IDO processing procedure commands are: JECHO, JFRCERR, JNAV, and JSOMECMD.

Command Name and Abstract:

IERR - This is the IDO error handler.

Command Format:

IERR type arg1

(arg1 is the error type string)

Command Description:

This IDO command is the error handler which handles the following error types:

- failure in processing an IDO command file procedure. The IDO will attempt to recover by sending the IDO to another node (INAV).
- failure to receive an acknowledgement from a node receiving an IDO - No recovery action is attempted.
- default - No recovery action is taken.

Command Name and Abstract:

IEXPND - Expands the IDO CPIO file.

Command Format:

IEXPND

Command Description:

This IDO command expands the IDO CPIO file within the appropriate environment to enable processing to be initiated.

Command Name and Abstract:

IJOB - Executes all processing procedures sequentially listed in the IDO ICMD file.

Command Format:

IJOB

Command Description:

This IDO command will loop through the IDO ICMD file using an index to maintain position of the next job to execute (INCRF). IVALID is called to verify if the command is valid for this node. ICKHALT is checked to see if the processing has been halted. ILOG is updated as the processing commands are performed.

Command Name and Abstract:

ILOG - This is the IDO logging handler.

Command Format:

ILOG arg1

(arg1 is the string to be inserted as an entry in the log file)

Command Description:

This IDO command will insert a log entry into the log file with the following format:

[current date] [node name] [IDO name] [string entry]

Command Name and Abstract:

INAV - This command is a simple stub to simulate navigation to another node after an error condition has occurred.

Command Format:

INAV

Command Description:

This is a stub command used to demonstrate when the IDO attempts to execute a job in ICMD that is either not allowed or is not available at this node. This is "not" the same as the intentional navigation which is performed by JNAV.

Command Name and Abstract:

INCRF - Used to increment the index which keeps track of the next command in the ICMD file.

Command Format:

INCRF

Command Description:

INCRF is an IDO command which is used to index the ICMD file. This is necessary because UNIX shell does not support indexed files.

Command Name and Abstract:

IPROC - Executes the IDO processing procedures.

Command Format:

IPROC

Command Description:

This IDO command executes the IDO processing procedures of an IDO. The processing is transferred to the correct IDO environment, where the IDO CPIO file is expanded back to its components when IEXPND is called. The command IACK will return an acknowledgement to the sending node that the IDO was received. IJOB is called to process any procedures/commands contained in ICMD. ILOG and IERR are called appropriately during each IDO processing procedure. Final / ICLEAN is called to clean up the IDO directory when all job processing is completed.

Command Name and Abstract:

ISEND - Sends an IDO to a node.

Command Format:

ISEND arg1 arg2

(arg1 is the node the file is to be sent to)

(arg2 is the file name of the IDO to be sent)

Command Description:

The named file is moved to the directory RJEOUT and then transmitted to the named node.

Command Name and Abstract:

IVALID - This is a simple stub command used to simulate the manager verifying if a job is valid for the local node.

Command Format:

IVALID arg1

(arg1 is job name)

Command Description:

This is a simple stub used to facilitate implementation. It is used to simulate calling an IDO processing procedure which determines if a job is valid for the local node.

Command Name and Abstract:

IWACK - Waits for an acknowledgement from an IDO which was sent to another node.

Command Format:

IWACK

Command Description:

This IDO command waits for an acknowledgement to be sent to the local RJEIN directory from an IDO which has been sent to another node. When an acknowledgement is received, an entry is made in the IDO log (ILOG). An error message is entered into the log file if the acknowledgement takes too long.

Command Name and Abstract:

JECHO - This command is a stub which makes just a log file entry.

Command Format:

JECHO arg1

(arg1 is a string)

Command Description:

This command passes to ILOG the command name "JECHO" and a string argument. It is used as a stub to demonstrate that this IDO processing command was indexed and executed.

Command Name and Abstract:

JFRCERR - This command forces an error condition, and makes and IDO log entry.

Command Format:

JFRCERR arg1

(arg1 is a string)

Command Description:

This command passes to ILOG a log entry which contains the name "JERR" and a string argument. A failure is forced and return is made to the calling procedure. The command is used as a stub to demonstrate that this IDO processing command was indexed and executed.

Command Name and Abstract:

JNAV - This command navigates an IDO to another node.

Command Format:

JNAV arg1 arg2

(arg1 is the name of the node the file is to be sent to)

(arg2 is "yes" if the node manager is to wait for an acknowledgement -or- "no" if the node manager is NOT to wait for an acknowledgement)

Command Description:

This command passes ILOG a log entry that an IDO is being sent to another node. The IDO file is constructed using IBUILD and routed using ISEND. After transmittal IWACK waits for an acknowledgement sent from the receiving node.

Command Name and Abstract:

JSOMECMD - This command is a stub which makes just a log file entry.

Command Format:

JSOMECMD arg1

(arg1 is a string)

Command Description:

This command passes to ILOG a log entry the command name and a string argument. It is used as a stub to demonstrate that this IDO processing command was indexed and executed.

Command Name and Abstract:

NINIT - Initialize one node structure.

Command Format:

NINIT arg1

(arg1 is the name of the node)

Command Description:

This node command will create a node structure by making a directory named after the argument passed with the command. Additional directories will be created underneath the node directory of: bin, RJEIN, RJEOUT, LOG, MISC, and DATA.

After all the directories are created the command will then initialize the "bin" directory with all the I-commands and all the N-commands..

Command Name and Abstract:

NLOG - The logging handler for each node.

Command Format:

NLOG arg1

(arg1 is a string entry for the node log file)

Command Description:

This node command installs string entries into the individual node log files.

Command Name and Abstract:

NMAIN - Executes the main node module.

Command Format:

NMAIN

Command Description:

This node command executes on a scheduled basis two main tasks:

- scheduling of incoming IDO's to be processed
- sending waiting outgoing IDO's via execution of the module IPROC and also installs appropriate processing information into NLOG

Command Name and Abstract:

NSCHED - Schedules NMAIN at each individual node to run periodically.

Command Format:

NSCHED arg1

(arg1 is the node name)

Command Description:

This node command sets up the local node environment and calls the ICKHALT module to see if simulation has been halted. NMAIN is then called to execute the main program module on a scheduled periodic basis.

Command Name and Abstract:

SBUILD - Builds the IDO CPIO file.

Command Format:

SBUILD

Command Description:

This simulation command calls the IBUILD module.

Command Name and Abstract:

SCLEAN - Cleans up the specified node(s).

Command Format:

SCLEAN arg1 arg2 ...

(arg1 arg2 ... are the specific names of the node(s) to be cleaned up)

Command Description:

This simulation command is executed simulation of the IDO implementation to cleanup (remove all files etc.) under the specified node.

Command Name and Abstract:

SETENV - Sets up the operator environment for entering an IDO.

Command Format:

SETENV arg1 arg2

(arg1 is the node name)

(arg2 is the IDO name)

Command Description:

This simulation command must be executed with a '.' to work.

You must have already run SINIT before using SETENV. Normally the node used has been created by SINIT, but will NOT be started by SSTARTUP. This enables you to manually control what happens at this node.

If the IDO does not yet exist under the selected node, it will be created as well as some empty working files. The user can hand edit the files to create the IDO or copy them in from somewhere outside of the IDO simulation directory structure. Once the IDO files are created use SBUILD to build a CPIO file and execute SSEND

to send it to a node.

Command Name and Abstract:

SHALT - Temporarily halts the IDO simulation.

Command Format:

SHALT

Command Description:

This simulation command is entered manually by the user during the IDO implementation simulation to temporarily halt the simulation. This will allow the user to examine log files, error files, etc. during the simulation.

IDO processing procedures will automatically restart processing or else execution of the SRESUME command will restart processing.

Command Name and Abstract:

SINIT - Initializes simulation directories.

Command Format:

SINIT arg1 arg2 ...

(arg1 arg2 ... are the names of the nodes to be used during the simulation)

Command Description:

This command sets up the environment for the IDO simulation and also initializes the simulation directories (NINIT) based on the number of nodes requested via the command arguments.

Command Name and Abstract:

SKILL - Stops all simulation.

Command Format:

SKILL

Command Description:

This simulation command is entered manually by the user during simulation of the IDO implementation to stop all simulation.

Command Name and Abstract:

SRESUME - Allows simulation execution to resume once it has been temporarily halted.

Command Format:

SRESUME

Command Description:

This simulation command is entered manually by the user during simulation of the IDO implementation to allow the simulation execution to resume once it has been temporarily halted.

Command Name and Abstract:

SRUNSIM - Performs a complete simulation run, except for termination.

Command Format:

SRUNSIM

Command Description:

This is the master simulation command which initiates the setup and running of the simulation. All traces of any previous simulation are removed as well as any IDO directory structures. A new IDO directory structure is created. An IDOSAV directory must exist with a copy of all the commands, modules, etc., which this command uses to copy the files into the working simulation directories. This command must be executed with '.' to be able to setup the simulation environment, which will allows the user to go in and manually alter the IDO files.

Command Name and Abstract:

SSEND - Sends an IDO to another node.

Command Format:

SSEND arg1 arg2

(arg1 is the node the IDO is to be sent to)

(arg2 is the name of the IDO to be sent)

Command Description:

This simulation command will send the named IDO to the RJEIN directory of the named node.

Command Name and Abstract:

SSTARTUP - Starts up the node(s) for the simulation.

Command Format:

SSTARTUP arg1 arg2 ...

(arg1 arg2 ... are the names of the nodes the user wants to be started up)

Command Description:

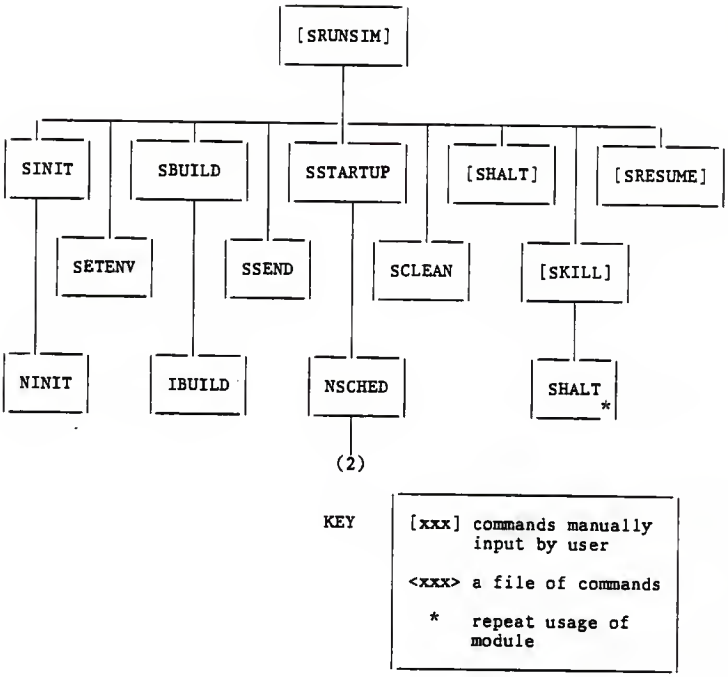
This simulation command will set up the simulation environment and then for each node named start the simulation based upon the IDO processing procedures in NSCHED.

APPENDIX II - SOURCE CODE STRUCTURE CHART

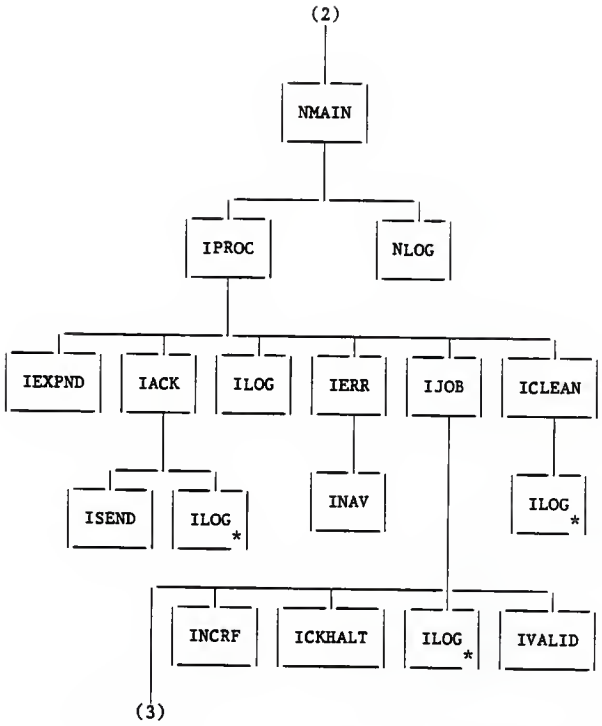
This appendix contains a hierarchical structure chart of the source code modules.

APPENDIX II

SOURCE CODE
HIERARCHICAL STRUCTURE CHART



SOURCE CODE
HIERARCHICAL STRUCTURE CHART

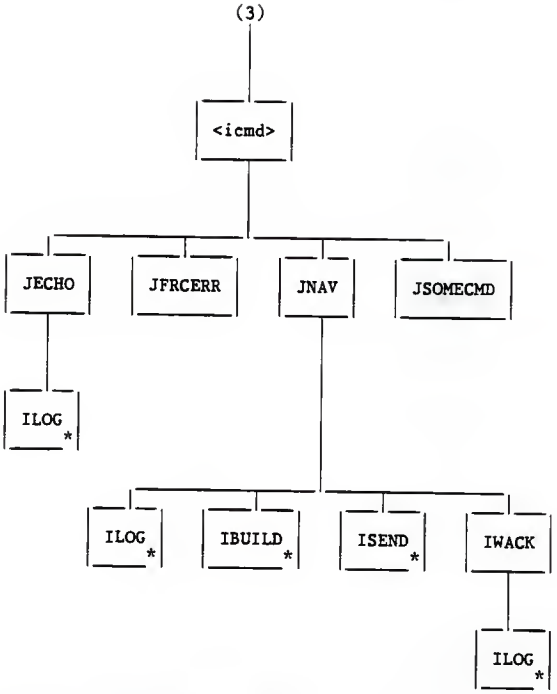


KEY

[xxx] commands manually input by user

<xxx> a file of commands

* repeat usage of module



KEY

[xxx] commands manually input by user

<xxx> a file of commands

* repeat usage of module

APPENDIX III - SOURCE CODE LISTINGS

This appendix contains the source code listings for the simulation of the "Management Of An Intelligent Data Object" implementation project.

CMD ARGS	DESCRIPTION
IACK	Acknowledge receipt of IDO
IBUILD	Build IDO file from directory into cpio file
ICKHALT	Check if simulation halted, wait until running
ICLEAN	Clean up IDO directory and files
icmd (file)	File which has IDO processing procedure command list
IERR errtype string	Error handler
IEXPND	Expand IDO file into directory
IJOB	Execute jobs in IDO icmd file
ILOG string	Log IDO progress into IDO log file
INAV jobname	Navigate to process IDO at another node
INCRF string	Increment numeric value of string
IPROC	Process one IDO
ISEND node file	Send file to another node
IVALID jobname	Check if a procedure in icmd in IDO is valid
IWACK	Wait for acknowledgement
JECHO args	Echo args to IDO log file
JFRCERR args	Echo args and force failure
JNAV node ack	Procedure to navigate to another node - 'ack' is yes to wait for ack
JSOMECMD args	Echo args to IDO log file
NINIT node	Build node directories
NLOG string	Enter string into node log
NMAIN	Main program for a node
NSCHED	Schedule main node program
SBUILD	Build IDO .cpio file

SCLEAN node	Clean up node files, directories
SHALT	Temporarily halt simulation
SINIT node1 n2 ...	Initialize simulation nodes
SKILL	Terminate simulation node processes
SRESUME	Start a halted simulation
SRUNSIM	Run a complete simulation with a sample IDO
SSEND node	Send IDO .cpio file to node
. SSETENV node idname	Set up environment for node and IDO name idname
SSTARTUP node1 n2 ...	Startup simulation nodes

NOTES:

I = IDO

S = simulation

N = node

J = procedure/job

```
# Send acknowledgement for a received IDO

cd $IDOHOME

# Determine who sent us IDO

FROM=`cat from`

# Construct acknowledgement file to send

echo "$NODE received $IDO from $FROM" > $IDO.ack

# Send file to who sent us the IDO

ISEND $FROM $IDO.ack
ILOG "Ack sent to $FROM"
```

```
# Collect files into cpio format

cd $IDOHOME

# Remove any old file with same name

rm -f $IDO.cpio

# Make sure from file points back to this node

echo $NODE > from

# Create new cpio file in parent directory to avoid
# endless loop if cpio tries to cpio its own output file.

find . -print | cpio -oc > ../$IDO.cpio 2>/dev/null

# Move new file here where it belongs

mv ../$IDO.cpio .
```

```
# Check if simulation halted
# The file "sshalt" should contain the string
# "run", "halt" or "kill".

# Wait in a loop if halted
while true
        do
        case `cat $SIMHOME/sshalt` in
        halt)   #Simulation halted - sleep 10 sec. and
                #wait for resume or kill
                sleep 10
                continue
                ;;
        kill)   #Simulation terminated.  Since this command
                #was executed with '.' exit will force return
                #of the calling command.
                echo Terminating NODE=$NODE
                EXVAL=2
                break
                ;;
        *)      #default - treat as run.
                #break out of while loop and resume execution
                #of calling command
                EXVAL=0
                break
                ;;
        esac
        done
if test ! $EXVAL = 0
        # Force calling command to exit
        then exit $EXVAL
        fi

# Fall through to continue execution of caller
```

```
# Remove IDO directory

cd $IDOHOME

# Log this step

ILOG "ICLEAN removing directory"

# Save log/error files

mv ilog $NODEHOME/log/$IDO.ilog

cd $NODEHOME
rm -rf $IDO
```

```
# This is a sample IDO procedure file with sample commands

JECHO "Executing first command"
JECHO "Executing 2nd command"
JNAV b yes
JECHO "Executing 4th command"
JSOMECMD "Executing 5th command"
JNAV op no
JSOMECMD "Executing 7th command"
```

```
#
# This is the IDO error handler.
#
# Arg 1 is error type, additional args depend on type
#
# This is a very dumb error handler.  Much more
# sophistication could be added.
#

ILOG "IERR type $1"

case $1 in
        joberr) #Failure in processing cmd file job.
        INAV    #Attempt to recover by sending IDO to
                #another node.
        ;;
        nack)   #Failed to receive acknowledgement from
                #node receiving IDO.
        ;;
        *)      #default
        ;;
esac
```

```
# Expand cpio file

cd $IDOHOME
cat $IDO.cpio | cpio -icumd 2>/dev/null
```

```
# Process an IDO processing procedure in "icmd" file

# Set PATH to access local IDO supplied procedure commands

PATH=$IDOHOME/bin$PATH

INDEX=0                 # Local index through cmd file
CMDIX=`cat icmdinx`     # IDO index to last procedure executed.
                        # Procedures start with number 1.
                        #   0 is null.

# Loop through procedures in command file

{
while read JOBNAME JOBARGS
        do

        # Skip comment lines in "icmd" starting with '# '
        if test $JOBNAME = '#'
                then continue
        fi

        # Increment index

        INDEX=`INCRF $INDEX`

        # Skip procedures up to next procedure to be executed.
        # This is brute force because UNIX shell
        # doesn't support indexed files.
        #
        # Ideally, we would just start with the next job
        # after the last one successfully executed.

        if test $INDEX -le $CMDIX
                then

                # Not there yet, continue while loop.

                continue
        fi

        # Check validity of this procedure at this node.

        IVALID $JOBNAME $JOBARGS
        RET=$?  # Set RET to the return from IVALID
        if test $RET != 0
                then

                # All done with procedures at this node.

                exit $RET
        fi
```

```
        # Index points to the procedure to execute - save it.

        echo $INDEX > icmdinx

        # Save procedure attempting to execute

        echo $JOBNAME > lastcmd

        . ICKHALT       # Check if halted - exits if killed
        ILOG "IJOB processing procedure $INDEX $JOBNAME"

        # Execute procedure

        "$JOBNAME" "$JOBARGS"
        RET=$?          # Set RET to the return code
                        # from procedure.
        if test $RET = 0

                # Successful procedure

                then
                continue
        else

                # Procedure failed or completed at this node

                exit $RET       # Failure return -
                                # pass back procedure ret.
        fi
        done
} < icmd

# Completed all procedures in IDO procedure file for this node.

exit 0
```

```
#
# Logging handler which will log each procedure during
# processing of an IDO into the IDO log file.
#
# Arg 1 is a string for the log file entry
#
# Echo date and time, node identity, and IDO identity
# as well as requested message
#

echo "`date` NODE=$NODE IDO=$IDO $1" >> $IDOHOME/ilog
```

```
#
# Navigate to another node to continue processing.
#
# This is used when an IDO attempts to execute a procedure
# in "icmd" that is not allowed or is not available at this
# node.  Note this is not to be confused with the intentional
# navigation done by the JNAV command.
#
# This command needs to be much smarter and be able to
# determine where to send the IDO to best handle its
# its current (error) condition.
#
# This is a "stub" that does nothing, except prove that
# the processing got here.
#
# A possible extension might be to:
#    Look at 1st command attempted.
#    Look it up in a table to see which node
#      should handle this type of failure.
#    Do IBUILD, ISEND, IWACK, ICLEAN ...
#      to pass the IDO along.
#
# This function is not really needed here if IVALID is
# implemented, except possibly as a means to retry certain
# types of commands that failed.
#
```

```
#
# This module is used to:
#
# Increment positive (>=0) string argument and return string.
#
# This is necessary because shell doesn't know how to do this.
#

dc <<!
$1 1 + p !
```

```
# Process an IDO

cd $IDOHOME

# Expand IDO

IEXPND

# Send acknowledgement

IACK
ILOG "Processing started in IPROC"

# Process procedure in "icmd" file
IJOB
RTRN=$?         # Set rtrn to return code from IJOB
case $RTRN in

        0)      #Success
                ILOG "IJOB COMPLETED"
                ;;

        1)      #Failure of command
                IERR joberr
                ;;

        2)      # Simulation terminated
                # Leave node intact and return.
                exit 0
                ;;

        3)      # Failure to send IDO to another node.
                IERR nack
                ;;

        4)      # IDO processing navigated to another node
                ILOG "IJOB NAVIGATED"
                ;;

        *)      #Default
                ILOG "IJOB UNKNOWN COMPLETION CODE"
                ;;
esac

# Cleanup, remove directory

ICLEAN $1
```

```
#
# Send file to a node
#
# Arg1 is node to send to and arg2 is file name
#
# Move file to rje for transmission
#

cp $2 $NODEHOME/rjeout

# Remove any possible old acknowledgements

rm -f $NODEHOME/rjein/$2.ack

# Create a file containing destination
echo $1 $2 > $NODEHOME/rjeout/$2.dest
```

```
#
# Determine if a IDO job is valid for this node
#
# Arg 1 is job name, additional args are procedure args
#
# This command should look at the procedure name and
# decide if that procedure can/should be executed at
# this node.  It could then navigate, etc.
#
# Also reference INAV.
#
# Like INAV, this is a simple "stub" to demonstrate that
# this module was reached.
#
```

```
# Wait for acknowledgement from an IDO sent to another node

ILOG "Waiting for ack"

LOOPCT=0
while test $LOOPCT -lt 10
        do
        sleep 20
        if test -s $NODEHOME/rjein/$IDO.ack
                then
                ILOG "Ack received"
                rm $NODEHOME/rjein/$IDO.ack
                exit 0
        fi

        #Increment count
        LOOPCT=`INCRF $LOOPCT`
done

ILOG "Timed out waiting on ack"
exit 3
```

```
#
# This is a sample IDO processing procedure stub
#
# Echo command name and a string (all args) to log file
#

ILOG "JECHO $*"
```

```
#
# This is a sample IDO processing procedure stub
#
# Procedure to force error
# All arguments are logged to IDO log and fail returned
#

ILOG "JERR $*"
exit 1
```

```
#
# This is a sample IDO processing procedure stub
#
# Procedure to navigate to another node
# Arg 1 is node
# Arg 2 is "yes" to wait for an acknowledgement,
#   or "no" for no wait for an acknowledgement
#

cd $IDOHOME

ILOG "JNAV navigating to node $1"

# Build IDO file

IBUILD

# Send to node

ISEND $1 $IDO.cpio

if test "$2" != yes
        then
        # No ack requested
        exit 4
        fi

# Wait for acknowledgement

if IWACK
        then
        exit 4  # Successful navigation
else
        exit 3  # Signal acknowledgement failure
fi
```

```
#
# This is a sample IDO processing procedure stub
#
# Echo command to IDO log file
# All args are echoed
#

ILOG "JSOMECMD $*"
```

```
#
# Initialize one node
# Arg is node name
#

cd $SIMHOME
NODE=$1
mkdir $NODE
cd $NODE
mkdir bin rjein rjeout log misc data

#
# Copy IDO handling commands to node bin.
#
# This is optional in this simulation since
# the $PATH includes $SIMHOME/bin, but
# it would be necessary to have them there
# in the real world where $SIMHOME wouldn't exist.
#

cp $SIMHOME/bin/I* bin
cp $SIMHOME/bin/N* bin
```

```
#
# Logging Handler for node
#
# Arg 1 is a string for node log file entry
#

echo "`date` NODE=$NODE $1" >> $NODEHOME/log/log
```

```
#
# Main execution entered to run periodically.
#
# Performs two main tasks: scheduling incoming IDOs to
# be processed, and to send waiting outgoing files/IDOs.
#

# Schedule incoming jobs

export NODE NODEHOME IDO IDOHOME
cd $NODEHOME/rjein

# IDLIST is a string consisting of all the .cpio file names

IDLIST=`ls *.cpio 2>/dev/null`
cd $NODEHOME

# Loop through all files in list.

for IDFILE in $IDLIST
        do

        # Strip .cpio off file name to get IDO name
        IDO=`echo $IDFILE | sed 's/.cpio//'`
        IDOHOME=$NODEHOME/$IDO

        # Make a directory for IDO
        mkdir $IDO
        mv $NODEHOME/rjein/$IDFILE $IDOHOME

        # Schedule independent background process for IDO.
        IPROC &
        NLOG "$IDO received and scheduled by NMAIN"

        done

# Handle waiting outgoing IDOs and files

cd $NODEHOME/rjeout
for FNAME in `ls *.dest 2>/dev/null`
        do
        { read DEST FILE < $FNAME
        mv $FILE $SIMHOME/$DEST/rjein
        rm $FNAME
        NLOG "FILE $FILE sent to DEST $DEST"
        } < $FNAME
        done
```

```
#
# Schedule main program at a node to run periodically
#
# Arg 1 is node name to start up
#

# Setup environment

export SIMHOME NODE NODEHOME IDO IDOHOME PATH
NODE=$1
NODEHOME=$SIMHOME/$NODE
PATH=:$NODEHOME/bin$PATH
rm -f $NODEHOME/log/*

while true
        do
        . ICKHALT       # Check if simulation halted

        #echo Waking up node $NODE
        . NMAIN         # Execute main node program
        sleep 20        # Wait before next entry
        done
```

```
#
# Build IDO cpio file
#
   . IBUILD
```

```
#
# Clean up nodes
# Args are node names
#

cd $SIMHOME
for NODE in $*
        do
        rm -rf $NODE
        done
```

```
#
# Command to set up operator envirnment for entering an IDO.
#
# Note this command must be executed with '.' to work.
#
# It chooses an operator node named "op" and a test IDO name
# of "i01".
#
# You must have already run SINIT before using SETENV,
# because it creates the "op" node.
#
# Normally the op node used has been created by SINIT but will
# NOT be started by SSTARTUP.  This is so that you can manually
# control what happens at this node without the node software
# interfering and trying to do its thing.
#
# If the IDO does not yet exist under the selected node, it
# will be created along with some empty files.  Then, you
# would hand edit the files to create the IDO or copy them in
# from somewhere outside the IDO simulation directory (idosav).
# Once you have the IDO files the way you want them, use
# SBUILD to build a cpio file and SSEND to send it to a node.
#

# Setup environment

export SIMHOME NODE NODEHOME IDO IDOHOME PATH
SIMHOME=$HOME/ido
NODE=op
IDO=i01
NODEHOME=$SIMHOME/$NODE
IDOHOME=$NODEHOME/$IDO

# Add IDO simulator bin if not already in path.
# Search PATH to see if "ido" is already there.
# NOTE: <<!....! takes stdin to be ... .

if grep ido > /dev/null 2>&1 <<!
$PATH
!
        then
                # already in path
                true #dummy cmd to keep shell happy
        else
                # Note if PATH does not begin with ':' then
                # a ':' is needed before $PATH
                PATH=:$SIMHOME/bin$PATH
        fi

# You can turn echoes on if you want to show path established
#echo NODE=$NODE
#echo NODEHOME=$NODEHOME
```

```
#echo IDO=$IDO
#echo IDOHOME=$IDOHOME
#echo PATH=$PATH

# Create sample IDO directory if it doesn't already exist.
cd $NODEHOME
if test ! -d "$IDO"
        then
        mkdir $IDO
        cd $IDO
        mkdir bin data
        echo > icmd
        echo 0 > icmdinx
        echo $NODE > from
        echo > ilog
        echo > lastcmd
        fi
cd $IDOHOME
echo "You are in $IDOHOME"
```

```
#
# Temporarily halt simulation
#

echo "halt" > $SIMHOME/sshalt
```

```
#
# Initialize simulation directories
# Args are names of nodes to use
#

# Set up environment

export IDO IDOHOME NODE NODEHOME SIMHOME
SIMHOME=$HOME/ido
cd $SIMHOME
PATH=:$SIMHOME/bin$PATH

# Initialize nodes
for node in "$*"
        do
        NINIT "$NODE"
        done
```

```
#
# Stop simulation
#

echo "kill" > $SIMHOME/sshalt
```

```
#
# Resume simulation execution
#

echo "run" > $SIMHOME/sshalt
```

```
#
# This file is an attempt to setup and run a complete
# simulation with just one command.
# It has no arguments and makes a lot of assumptions.
#
# It first kills all processes spawned by this user/terminal.
#       NOTE THIS COULD BE VERY NASTY !!
# You can also cleanup in this same way using 'kill 0'.
#
# It removes all traces of any previous simulation and
# the IDO directory.
# It expects an "idosav" directory with a copy of all the
# commands, etc.
# $HOME/idosav and subtending directories/files is all
# that is needed.
#
# For the SHALT, SRESUME, and SKILL commands to work, this
# SRUNSIM command must be executed with '.' to set up your
# environment.
#

echo "Killing all background processes"
kill 0

cd $HOME
echo "Removing IDO directory and associated junk"
rm -rf ido

echo "Creating new IDO directory"
mkdir ido
cd ido

echo "Creating ido/bin and copying over all commands"
mkdir bin
cp $HOME/idosav/bin/* bin
cd bin

echo "Setting up three nodes: a, b, and op"
SINIT a b op

echo "Setting up operator environment for node op, ido i01"

# We should be in $HOME/ido/op/i01
# To change sample IDO, edit the stuff in $HOME/idosav/op/i01

echo "Copying sample IDO to op node"

# Note this is a kludge, should really use SBUILD.
# The problem is that SBUILD looks at the environment and
# will build what is in ido/op/i01, not idosav/i01.
#
# This could be avoided by instead copying the idosav/i01
```

```
# node to ido/op/i01 and then using SBUILD.

cd $HOME/idosav/i01

# ****NOTE**** there cannot be a i01.cpio file in idosav/i01!

rm -f *cpio

# cpio sample IDO into one file and put in op node.

find . -print | cpio -oc > $HOME/ido/op/i01/i01.cpio 2>/dev/null

# Expand it so when you look around, something is there

cd $HOME/ido/op/i01
cat i01.cpio | cpio -icumd 2>/dev/null

echo "Sending i01.cpio file to node a"
echo "Nothing will happen until we startup the nodes"

SSEND a

echo
echo "It will take a minute or two to process an"
echo "IDO depending on how many nodes it visits."
echo
echo "The sample being executed will start at node"
echo "a, then go to b, then back to op."
echo
echo "After the nodes are started up --- "
echo "Enter one of the following:"
echo "  SHALT to halt the simulation"
echo "  SRESUME to resume execution"
echo "  SKILL to kill all the simulation processes"
echo "  'kill 0' if some processes won't go away"
echo
echo "Starting up nodes a and b"

SSTARTUP a b
sleep 1

echo "You are on your own from here."
```

```
#
# Send a file to another node
# Arg 1 is node, arg 2 is file
#

cp $IDO.cpio $SIMHOME/$1/rjein
```

```
#
# Startup simulator
# Args are nodes to start
#

# Set up environment

export IDO IDOHOME NODE NODEHOME SIMHOME
SIMHOME=$HOME/ido
cd $SIMHOME
PATH=:$SIMHOME/bin$PATH
echo run > $SIMHOME/sshalt

# Start up independent background process for each node

for NODE in $*
        do
        NSCHED $NODE&
        # Delay between nodes to reduce peak process load
        sleep 10
        done
```

APPENDIX IV - SAMPLE LOG FILES

This appendix contains samples of node log files and IDO log files as they appear at the end of a complete simulation of the node manager implementation.

NODE LOG FILE OF NODE "a"

```
Sat Jun 29 13:34:21 CDT 1985 NODE=a i01 received and scheduled
   by NMAIN
Sat Jun 29 13:34:43 CDT 1985 NODE=a FILE i01.ack sent to DEST op
Sat Jun 29 13:34:44 CDT 1985 NODE=a FILE i01.cpio sent to DEST b
```

IDO LOG FILE AT NODE "a"

```
Sat Jun 29 13:34:25 CDT 1985 NODE=a IDO=i01 Ack sent to op
Sat Jun 29 13:34:25 CDT 1985 NODE=a IDO=i01 Processing
   started in IPROC
Sat Jun 29 13:34:29 CDT 1985 NODE=a IDO=i01 IJOB processing
   job JECHO
Sat Jun 29 13:34:30 CDT 1985 NODE=a IDO=i01 JECHO "Executing
   first command"
Sat Jun 29 13:34:31 CDT 1985 NODE=a IDO=i01 IJOB processing
   job JECHO
Sat Jun 29 13:34:32 CDT 1985 NODE=a IDO=i01 JECHO "Executing
   2nd command"
Sat Jun 29 13:34:33 CDT 1985 NODE=a IDO=i01 IJOB processing
   job JNAV
Sat Jun 29 13:34:34 CDT 1985 NODE=a IDO=i01 JNAV navigating
   to node b
Sat Jun 29 13:34:37 CDT 1985 NODE=a IDO=i01 Waiting for ack
Sat Jun 29 13:35:18 CDT 1985 NODE=a IDO=i01 Ack received
Sat Jun 29 13:35:19 CDT 1985 NODE=a IDO=i01 IJOB COMPLETED
Sat Jun 29 13:35:20 CDT 1985 NODE=a IDO=i01 ICLEAN removing
   directory
```

NODE LOG FILE OF NODE "b"

```
Sat Jun 29 13:34:52 CDT 1985 NODE=b i01 received and scheduled
   by NMAIN
Sat Jun 29 13:35:16 CDT 1985 NODE=b FILE i01.ack sent to DEST a
Sat Jun 29 13:35:19 CDT 1985 NODE=b FILE i01.cpio sent to DEST op
```

IDO LOG FILE AT NODE "b"

```
Sat Jun 29 13:34:25 CDT 1985 NODE=a IDO=i01 Ack sent to op
Sat Jun 29 13:34:25 CDT 1985 NODE=a IDO=i01 Processing
   started in IPROC
Sat Jun 29 13:34:29 CDT 1985 NODE=a IDO=i01 IJOB processing
   job JECHO
Sat Jun 29 13:34:30 CDT 1985 NODE=a IDO=i01 JECHO "Executing
   first command"
Sat Jun 29 13:34:31 CDT 1985 NODE=a IDO=i01 IJOB processing
   job JECHO
Sat Jun 29 13:34:32 CDT 1985 NODE=a IDO=i01 JECHO "Executing
   2nd command"
Sat Jun 29 13:34:33 CDT 1985 NODE=a IDO=i01 IJOB processing
   job JNAV
Sat Jun 29 13:34:34 CDT 1985 NODE=a IDO=i01 JNAV navigating
   to node b
Sat Jun 29 13:34:55 CDT 1985 NODE=b IDO=i01 Ack sent to a
Sat Jun 29 13:34:55 CDT 1985 NODE=b IDO=i01 Processing started
   in IPROC
Sat Jun 29 13:34:59 CDT 1985 NODE=b IDO=i01 IJOB processing
   job JECHO
Sat Jun 29 13:34:59 CDT 1985 NODE=b IDO=i01 JECHO "Executing
   4th command"
Sat Jun 29 13:35:02 CDT 1985 NODE=b IDO=i01 IJOB processing
```

```
   job JSOMECMD
Sat Jun 29 13:35:03 CDT 1985 NODE=b IDO=i01 JSOMECMD "Executing
   5th command"
Sat Jun 29 13:35:08 CDT 1985 NODE=b IDO=i01 IJOB processing
   job JNAV
Sat Jun 29 13:35:10 CDT 1985 NODE=b IDO=i01 JNAV navigating
   to node op
Sat Jun 29 13:35:14 CDT 1985 NODE=b IDO=i01 IJOB COMPLETED
Sat Jun 29 13:35:15 CDT 1985 NODE=b IDO=i01 ICLEAN removing
   directory
```

BIBLIOGRAPHY

[BAU80] Baumann, L.S. and Coop, R.D., "Automated Workflow Control: A Key To Office Productivity", Proc. AFIPS Office Automation Conf., Mar 1980, and Electronic Office Research Project, Sperry Univac, Roseville, Minn, National Computer Conference, 1980

[BYR82] Byrd, Roy J. and Smith, Stephen E. and deJong, S. Peter, "An Actor- Based Programming System", ACM 0-89791-075-3/82/006/0067, 1982

[COO80] Cook, Carolyn L., "Streamlining Office Procedures--An Analysis Using The Information Control Net Model", National Computer Conference, 1980

[DEU81] Deutsch, D., "Design Of A Message Format Standard", Computer Message System (R.sp2. Uhlig - editor), North-Holland Publishing Company, IFIP, 1981

[DIP83] DiPirro, J.E. and Ferrans, J.E. and Juszczak, C., "A Form Management System for Switching Database Administration", Proceedings IEEE International Conference On Communications, Boston, MA, June 19-22, 1983, pp. A4.1.1-A4.1.6 (pp. 125-130), Vol. 1

[ELL80] Ellis, Clarence A. and Nutt, Gary J., "Office Information Systems and Computer Science", Computing Surveys, Vol. 12, No. 1, March 1980

[ELL82] Ellis, Clarence A. and Bernal, Marc, "Officetalk-D: An Experimental Office Information System", ACM 0-89791-075-3/82/006/0131, 1982

[FER82] Ferrans, James C., "SEDL - A Language For Specifying Integrity Constraints On Office Forms", Proceedings ACM-SIGOA Conference On Office Information Skystems, Philadelphia, PA, June 21-23, 1982, pp. 123-130

[FIK81] Fikes, R.E., "Odyssey: A Knowledge-Based Assistant", Artificial Intelligence 16 (1981), pp. 331-361

[GAR81] Garcia-Luna, J.J. and Kuo, F.F., "Addressing And Directory Systems For Large Computer Mail Systems", Computer Message Systems (R. Uhlig - editor), North-Holland Publishing Company, IFIP, 1981

[GEH81] Gehani, N.H., "An Electronic Form System: An Experience In Prototyping", Bell Laboratories Research Report, June 1981

[GEH82] Gehani, Narain, "The Potential Of Forms In Office Automation", IEEE Transactions On Communications, Vol COM-30, No. 1, Jan 1982, pp. 120-125

[GEH83] Gehani, N.H., "High Level Form Definition In Office Information Systems", The Computer Journal, Vol. 26, No. 1, 1983

[GUT76] Guttag, J.V. and Horowitz, E. and Musser, D.R., "The Design Of Data Type Specifications", Current Trends In Programming Methodology, Vol IV, Data Structuring, Raymond T. Yeh - Editor, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

[HAM80] Hammer, Michael and Kunin, Jay S., "Design Principles Of An Office Specification Language", Proceedings AFIPS Office Automation Conference, Mar 1980, National Computer Conference

[HEW77] Hewitt, Carl and Baker, Henry Jr., "Actors and Continuous Functionals", Library For Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139, MIT/LCS/TR-194, December 1977

[HOG84] Hogg, John and Gamvroulas, Stelios, "An Active Mail System", Sigmod Record, Vol. 14, No. 2, 1984

[KER81] Kerr, I.H., "Interconnection Of Electronic Mail Systems - A Proposal On Naming, Addressing and Routing", Computer Message Systems (R.sp2. Uhlig - editor), North-Holland Publishing Company, IFIP, 1981

[KON82] Konsynski, Benn R. and Bracker, Lynne C. and Bracker, William E., "A Model For Specification Of Office Communications", IEEE Transactions On Communications, Vol. Com-30, No. 1, January 1982, pp. 27-36

[LAD80] Ladd, Ivor and Tsichritzis, D.C., "An Office Form Flow Model", Proceedings AFIPS Office Automation Conference, National Comp. Conf., Mar. 1980, University Of Toronto, Ont. Canada

[LEB82] Lebensold, J., and Radhakrishnan, T. and Jaworski, W.M., "A Modelling Tool for Office Information Systems", 1982 ACM 0-89791- 075-3/82/006/0141

[LIS74] Liskov, Barbara and Zilles, Stephen, "Programming With Abstract Data Types", SIGPLAN, April 1974

[MCB83] McBride, R. A. and Unger, E. A., "Modeling Jobs In A Distributed System", 1983 ACM 0-89791-123-7/83/012/0032

[MYL80] Mylopoulos, John and Bernstein, Philip A. and Wong, Harry K.T., "A Language Facility for Designing Database-Intensive Applications", ACM0362-5915/80/0600-0185, ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980, Pages 185-207

[SCH81] Schicker, P., "Service Definitions In A Computer Based Mail Environment", Computer Message Systems (R.sp2. Uhlig - editor), North-Holland Publishing Company, IFIP, 1981

[SCH82] Schicker, P., "Naming And Addressing In A Computer-Based Mail Environment", IEEE Trans. Commun., COM-30, 1 (Jan 1982), pp. 46-52

[SHO82] Shoch, "The Worm Programs--Early Experience With A Distributed Computation", CACM Vol 25, No. 3, March 1982

[STE80] Stefferud, Einar, "Electronic VS Paper Media Continua -- A Comparison", NCC '80 Personal Computing Digest

[STE81] Stefferud, E. and Mchugh, J., "The Role Of Computer Mail In Office Automation", Computer Message System (R.sp2. Uhlig - editor), North-Holland Publishing Company, IFIP,1981

[TAR81] Tarouco, L.M.R., "An Experimental Message Computer System Between Universities In Brazil", Computer Message Systems (R.sp2. Uhlig - editor), North-Holland Publishing Company, IFIP, 1981

[TSI80] Tsichritzis, D., "OFS: An Integrated Form Management System", Proceedings Of The ACM International Conference On Very Large Data Bases, 1980

[TSI81] Tsichritzis, D.C., "Integrating Database and Message Systems", Proc. 7th International Conference On Very Large Data Bases", 1981, pp. 356-362

[TSI82] Tsichritzis, D.C. and Rabitti, F.A. and Gibbs, S. and Nierstrasz, O.M. and Hogg, J., "A System For Managing Structured Messages", IEEE Trans. Commun., COM-30, 1(Jan 1982), pp. 66-73

[TSI82.1] Tsichritzis, D.C., "Forms Management", Commun ACM 25, 7(July 1982), pp. 453-478.

[TSI82.2] Tsichritzis, D. and Christodoulakis, S., "Message Files", ACM 0- 89791-075-3/82/006/0110, 1982

[TSI84] Tsichritzis, D., "Message Addressing Schemes", ACM Transactions on Office Information Systems, Vol. 2, No. 1, January 1984, Pages 58- 77

[VIT81] Vittal, John, "Active Message Processing: Messages As Messengers", North- Holland Publishing Company, IFIP, 1981

[VIT81.1] Vittal, J., "MSG-A Simple Message System", Computer Message Systems (R. Uhlig - editor), North-Holland Publishing Company, IFIP, 1981

[ZIS78] Zisman, Michael D., "Office Automation: Revolution or Evolution?", Sloan Management Review, Spring 1978

[ZLO81] Zloof, M.M., "QBE/OBE: A Language For Office And Business Automation", IEEE Computer (May 1981), pp. 13-22

ACKNOWLEDGEMENTS

I would like to thank Dr. Elizabeth A. Unger for serving as the Major Professor for my implementation project and this Masters Report. The numerous hours she spent in discussions, reviewing this report, and providing constructive criticism are very much appreciated.

I would also like to thank the other members of my committee, Dr. Virgil E. Wallentine and Dr. Richard A. McBride, for their interest in the project and report, even though their work load was quite heavy.

In addition, I appreciate the patient support and encouragement of CW (my husband), my project team members (the "Chicago 6"),[1] my friends, and my colleages.

The UNIX™ operating system environment was an invaluable aid. This report was edited using the VI screen editor, and formatted and printed using the MM Memorandum Macros text formatting package.

Last, but not least, without the existence of the AT&T Summer-On-Campus program, none of this would have been possible.

---

1. Sandy Bishop, Nancy Busack, Kathy Huml, Ronna Rykowski, and Rich Sewczwick

MANAGEMENT
OF AN
INTELLIGENT DATA OBJECT

by

DOROTHY MONTGOMERY GANTT

B.S., University of South Carolina, 1972

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

ABSTRACT

This report is based upon a team project developed by six students[1] working on their Masters reports at Kansas State University during the AT&T Summer-On-Campus program, Summer of 1985. The team project was based upon concepts in a paper, "Modeling Jobs In A Distributed System", written by R. A. McBride and E. A. Unger, in which an "intelligent data object" was described.

An Intelligent Data Object (IDO) may be defined as an instance of an intelligent abstract data type, which consists of a data structure and the set of operations defined on the object. The IDO also includes: routing information, a history, data, and specific processing/strategy information which logically resides within the IDO.

This report addresses the design of a "manager" which resides at each node. The manager will handle the receipt, processing, and sending of the IDO within the system. The manager will utilize the data contained within the IDO itself and the available resources resident at the node to perform all functions related to the IDO.

The model used for this implementation and simulation of this manager was a series of nodes analogous to multiple company locations, offices within a company, or terminals within a company linked through communication means to each other by physical and/or electronic means. Due to limitations of the available hardware to simulate and demonstrate the implementation of this project with independent processes at different nodes, independent processes were used running in different directories. Even though these directories did reside on a common machine with a common login, they communicated with each other through a single primitive which was the only communicating means between the processes in the multiple directory structures.

---

1. Sandy Bishop, Nancy Busack, Dottie Gantt, Kathy Huml, Ronna Rykowski, and Rich Sewczwick